ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE
TELECOMUNICACIÓN
UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Proyecto Fin de Carrera

Adaptación de servicios web tradicionales a Smartphones

AUTOR: Carlos Sánchez Berenguel
DIRECTOR: María Dolores Cano Baños
Septiembre / 2013

Índice

Índice de figuras

1 Introducción

La rápida penetración de los Smartphone, las inversiones en infraestructuras TIC y, en definitiva, la mejora en las prestaciones de los dispositivos móviles han contribuido al espectacular éxito de las aplicaciones para móviles. Así, se prevé que más de 73.000 millones de aplicaciones serán descargadas en todo el mundo en 2013. Estas aplicaciones han cambiado la forma en la que la información y el contenido online se presenta y se consume. Tradicionalmente, la mayoría de los proveedores de servicios creaban sus propias páginas web y atraían a clientes fieles que ayudaban a mantener su contenido en Internet. Sin embargo, como los dispositivos móviles inteligentes (Smartphones) se han vuelto más populares, los proveedores de servicios han comenzado a crear nuevas aplicaciones web optimizadas para dispositivos inteligentes mediante la reutilización del contenido online ya existente. A pesar los esfuerzos, proporcionar contenido online a los clientes móviles que utilizan las aplicaciones web sigue siendo difícil. Esto se debe principalmente a las limitaciones de la interacción entre aplicaciones y servicios web y las características de los dispositivos móviles inteligentes.

1.1 Objetivos del proyecto

El objetivo de este trabajo de investigación consiste en colaborar en el estudio del estado del arte sobre adaptación de servicios web tradicionales a Smartphones, incluyendo análisis de requerimientos, diseño de la arquitectura del sistema e implementación inicial de un caso práctico, con la posibilidad de incluir invocación dinámica de métodos de servicios web con intervención mínima del usuario. La aplicación se ha desarrollado para el sistema operativo iOS (iPhone OS), el sistema operativo desarrollado por Apple Inc.

1.2 Contenido

A continuación se va a comentar como acceden al contenido de internet los dispositivos móviles inteligentes y las tecnologías utilizadas. También se van a explicar las diferentes tendencias que existen en la actualidad a la hora de desarrollar aplicaciones para Smartphones, con sus ventajas e inconvenientes.

Además se explicará con detalle la aplicación que se ha desarrollado para el caso práctico, la tecnología empleada, los programas que han sido necesarios y la puesta en marcha para el desarrollo.

2 Acceso a los servicios web desde Smartphones

El modo en que nos relacionamos hoy con la tecnología es en gran medida debido a la revolución de las Apps, lo cual ha forzado una evolución de la web para adaptarse al ya, prácticamente consolidado entorno móvil [CAPT13].

El acceso a contenidos de internet se realizaba siempre desde un navegador web, pero con la salida al mercado de nuevos dispositivos móviles existen otras formas de acceder a la información. Por ejemplo, a través de aplicaciones nativas, entendiendo estas como aplicaciones que han sido desarrolladas de manera específica para un sistema operativo en concreto.

Cada año aumenta, de manera casi exponencial, el nº de internautas debido a la penetración de los Smartphones y las Tablets. Según las previsiones, para 2015, de los 3.5 millones de usuarios de internet previstos, 1.9 lo hagan desde dispositivos móviles, superando por primera vez la proporción de usuarios de móvil a los de escritorio, en este mismo 2013.

Con esta realidad, se ha desatado toda una revolución entre los desarrolladores para adaptarse a este nuevo entorno, adaptando los contenidos mediante varias tendencias, que sin dejar de ser incompatibles entre sí, tienen características y propósitos distintos.

2.1 Diseño web adaptable

Se trata de una técnica que permite que un mismo diseño y contenido Web se adapte a las pantallas de los dispositivos desde los que podemos ver dicha Web, incluso dependiendo de la orientación de la pantalla, ya sean Smartphones, tabletas, pantallas de ordenador e incluso la TV.

Con los nuevos dispositivos, cada uno con diferentes resoluciones de pantalla, que irrumpen en el mercado con regularidad, es fundamental que nuestro proyecto Web se adapte a ellos, de manera que el contenido sea líquido, ocupando todo el espacio disponible, y permitiendo así una correcta experiencia para el usuario, sin importar desde dónde se está accediendo. Esto se consigue con una única URL, un único contenido, código HTML5 y consultas CSS3.

Las ventajas son evidentes. Se consigue un ahorro en costes, no teniendo que hacer desarrollos paralelos para cada dispositivo. Ahorro en tiempo, ya que solo hay que realizar las actualizaciones de contenido una única vez para todos los entornos. Mayor soporte de navegadores.

2.2 Web Apps

Muy similar a lo anterior, encontramos esta otra tendencia, que siendo parecida en cuanto a la tecnología utilizada, comporta ciertas diferencias.

Básicamente se trata de una Web, totalmente adaptada a dispositivos móviles, que gracias a código HTML5, hojas de estilos CSS3 y JavaScript puede tener la misma apariencia que una App nativa, pero accesible vía navegador y que no necesita ser instalada en el dispositivo.

La principal ventaja es su versatilidad, al no ser descargada desde los markets, no es necesaria su validación. En caso de actualización, el usuario no tiene que volver a descargar, al tratarse de contenido web.

Como puntos en contra, decir que para este tipo de aplicaciones no es posible acceder a funciones del terminal como la geolocalización, la cámara o el acelerómetro. Pero esto no es un problema si lo que se busca, por ejemplo, es desarrollar una aplicación que facilita el acceso a contenidos, en la que si bien, el acceso a este tipo de funciones no es absolutamente imprescindible, los beneficios en cuanto a

ahorro en costes y plazos de implementación son más que evidentes, si lo comparamos con lo que supone desarrollar una App nativa para cada una de las principales plataformas (iOS, Android, Windows Phone, etc.) validarla en cada uno de los markets, mantenimiento, etc.

2.3 Aplicaciones nativas

En la actualidad las aplicaciones nativas son las favoritas del mercado, con 50 mil millones de descargas. Al estar específicamente diseñadas para el sistema, su rendimiento está optimizado y su interfaz concuerda con el aspecto del sistema en la mayoría de casos. Además, como forman parte de la tienda oficial de aplicaciones la visibilidad hacia el público está prácticamente garantizada y su seguridad es mayor.

Pero no todo son ventajas, una App nativa implica un control total de la aplicación por parte de la plataforma (que puede decidir si publicar la aplicación en su catálogo oficial o no siguiendo sus propias reglas), y en más de un caso estamos hablando de una plataforma cerrada.

2.4 Firefox OS

Hay algunos casos en los que lo nativo y lo basado en web se combinan, como puede ser el ejemplo de Firefox OS.

Firefox OS fue presentado en el Mobile World Congress de 2013 [FOS13], es un sistema operativo móvil desarrollado por Mozilla. Tiene un núcleo de Linux y un motor de renderizado basado en Gecko, que permite a los usuarios ejecutar aplicaciones desarrolladas por completo en HTML, JavaScript y otras APIs abiertas para aplicaciones web.

Una aplicación, mediante este sistema operativo, es una aplicación nativa del teléfono, pero al mismo tiempo también es una aplicación web. Se puede cargar momentáneamente, pero también se puede instalar para que quede guardada en el teléfono y se pueda usar sin conexión. Es decir, tiene las ventajas de los diferentes tipos de aplicaciones mencionadas anteriormente.

Según el responsable de Mozilla, está diseñado para que la web sea la alternativa a los muros cerrados de Apple y Google. La web permite una mayor flexibilidad, incluso la propia tienda de aplicaciones puede venir personalizada, optimizando las aplicaciones para un país o un mercado en concreto. Esta flexibilidad también se refleja en los desarrolladores, que pueden dejar de preocuparse en plataformas y en arquitecturas para desarrollar aplicaciones web que se adaptarán a todos los teléfonos con Firefox OS. Lo único que habrá que saber hacer es integrar esas aplicaciones web con los componentes del teléfono gracias al estándar HTML5.

Pero también tiene inconvenientes. Parece ser que Firefox OS no será, al menos en sus inicios, un sistema propio de teléfonos de alta gama. Firefox OS es algo bastante básico que necesitará recorrer un largo camino y demostrar lo que vale si quiere tener una comunidad de usuarios respetable.

3 Descripción de la aplicación

3.1 Diseño

3.1.1 Diseño de persistencia

Es necesario que la aplicación guarde los datos de las empresas introducidos por el usuario. De esta forma cada empresa queda registrada y el usuario puede acceder a la información directamente sin tener que volver a introducir los datos.

Los datos que se guardan de cada empresa son:

- El nombre de la empresa, de forma que el usuario pueda identificar las distintas empresas en la aplicación.
- La URL a la que se envía la petición para loguearse.
- El nombre de usuario y la contraseña requeridos para acceder a la cuenta del usuario en la empresa. Las claves se guardan cifradas, esto se verá con más detalle en el siguiente apartado.

3.1.2 Diseño de seguridad

En este apartado se va a tratar el almacenamiento de claves. Para una mayor seguridad se pretende guardar las claves cifradas. Se van a utilizar dos algoritmos de cifrado diferentes: el Algoritmo de *Hash* Seguro (SHA, *Secure Hash Algorithm*) y el algoritmo Estándar de Cifrado Avanzada (AES, *Advanced Encryption Standard*). El algoritmo SHA se usa para cifrar la clave del usuario, mientras que las contraseñas de las empresas registradas se cifran con el algoritmo AES usando la clave del usuario como clave para el cifrado y descifrado.

3.1.2.1 Algoritmo de Hash Seguro (SHA)

La familia SHA (*Secure Hash Algorithm*, Algoritmo de *Hash* Seguro) [SHA13] es un sistema de funciones *hash* criptográficas relacionadas de la Agencia de Seguridad Nacional de los Estados Unidos y publicadas por el *National Institute of Standards and Technology* (NIST). El primer miembro de la familia fue publicado en 1993 es oficialmente llamado SHA. Sin embargo, hoy día, no oficialmente se le llama SHA-0 para evitar confusiones con sus sucesores. Dos años más tarde el primer sucesor de SHA fue publicado con el nombre de SHA-1. Existen cuatro variantes más que se han publicado desde entonces cuyas diferencias se basan en un diseño algo modificado y rangos de salida incrementados: SHA-224, SHA-256, SHA-384, y SHA-512 (llamándose SHA-2 a todos ellos).

3.1.2.2 Algoritmo Estándar de Cifrado Avanzada (AES)

Advanced Encryption Standard (AES) [AES13], también conocido como Rijndael (pronunciado "Rain Doll" en inglés), es un esquema de cifrado por bloques adoptado como un estándar de cifrado por el gobierno de los Estados Unidos. El AES fue anunciado por el Instituto Nacional de Estándares y Tecnología (NIST) como FIPS PUB 197 de los Estados Unidos (FIPS 197) el 26 de noviembre de 2001 después de un proceso de estandarización que duró 5 años. Se transformó en un estándar efectivo el 26 de mayo de 2002. Desde 2006, el AES es uno de los algoritmos más populares usados en criptografía simétrica.

El cifrado fue desarrollado por dos criptólogos belgas, Joan Daemen y Vincent Rijmen, ambos estudiantes de la *Katholieke Universiteit Leuven*, y enviado al proceso de selección AES bajo el nombre "Rijndael".

3.2 Interfaz gráfica del usuario

Los principios de un buen diseño de una interfaz de usuario se basan en cómo piensa y trabaja el usuario [IHIG12], no en las capacidades del dispositivo. Una interfaz debe ser atractiva, intuitiva y lógica de manera que el usuario trabaje a gusto con la aplicación.

Además de tener una interfaz agradable, esta debe integrarse de forma adecuada con la funcionalidad de la aplicación. Es decir, deben de existir botones o controladores adecuados y debe de quedar claro para el usuario cual es la función que realizan.

También es importante la retroalimentación, que el usuario sepa en cada momento que es lo que está ocurriendo y el estado en que se encuentra la aplicación. Cuando realiza una acción espera una respuesta inmediata de la aplicación.

Según lo comentado se han definido las siguientes vistas que forman la aplicación. Se van a comentar a continuación cual es la función de cada una y que información proporcionan.

3.2.1 General

En las siguientes figuras se muestran las vistas de la aplicación y las relaciones entre ellas (véase Fig. 1 y Fig. 2). El controlador inicial es un *TabBarController* que proporciona la habilidad de cambiar entre diferentes vistas, dos en este caso. Muestra una tab bar en la parte inferior de la pantalla accesible desde cualquier lugar de la aplicación. La tab bar muestra diferentes pestañas definidas por un icono y una cadena de texto.

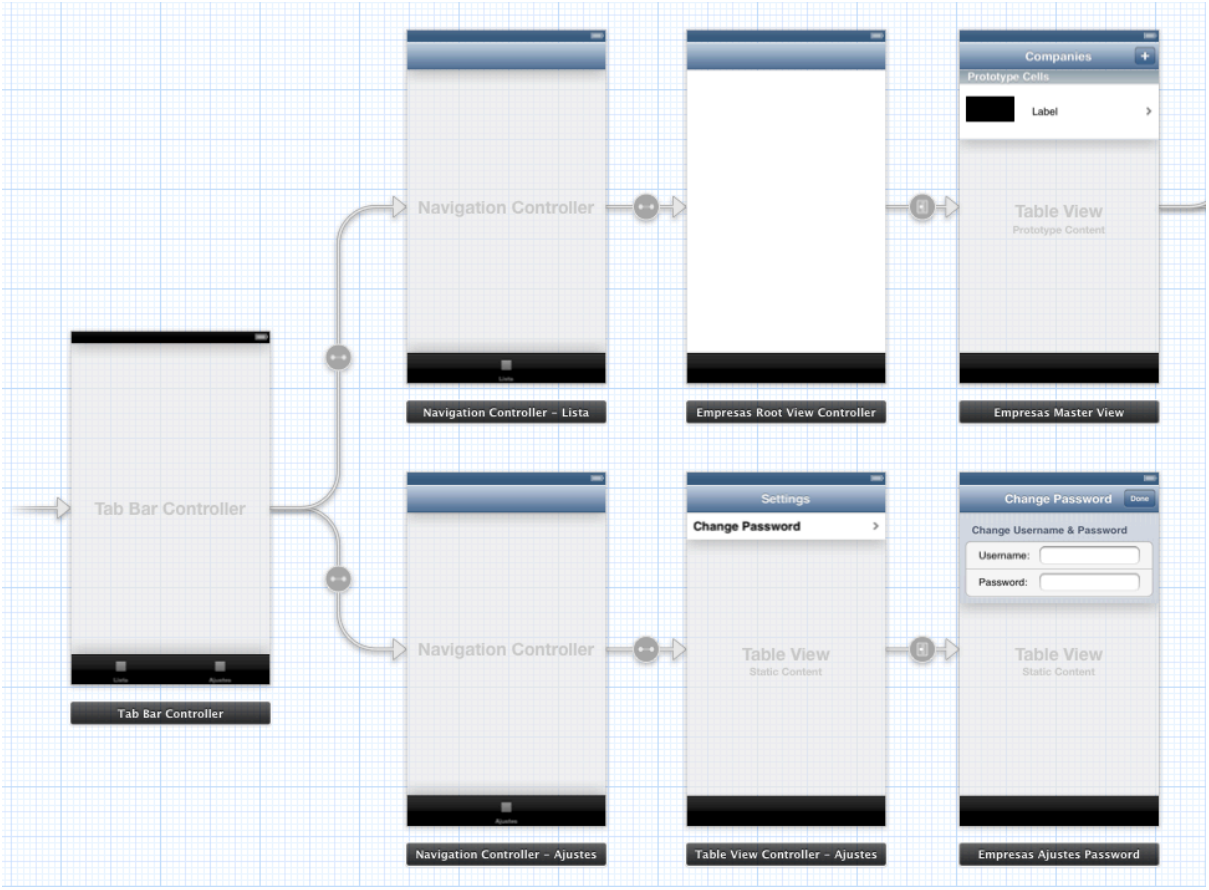

Figura 1. Diagrama de vistas

Cada una de las dos vistas contenidas en el *TabBarController* están contenidas a su vez en un *NavigationController*. Este controlador al igual que el anterior es un contenedor de controladores de vista que se usa para presentar los datos organizados de forma jerárquica. Aunque su función principal es la de administrar otros controladores de vista, también maneja unas pocas vistas. Principalmente gestiona una barra de navegación o *navigation bar*, que muestra información sobre la situación actual del usuario en la jerarquía de vistas. También proporciona un botón para volver a la vista anterior y otros controles que pueden ser necesarios para el usuario. Además el controlador de navegación puede controlar, de forma opcional, una barra de herramientas o *toolbar*, que se puede utilizar para mostrar diferentes acciones relacionadas con la vista actual.

Se puede observar como la aplicación parte de dos controladores de navegación y como el usuario puede ir avanzando en la jerarquía de las vistas.

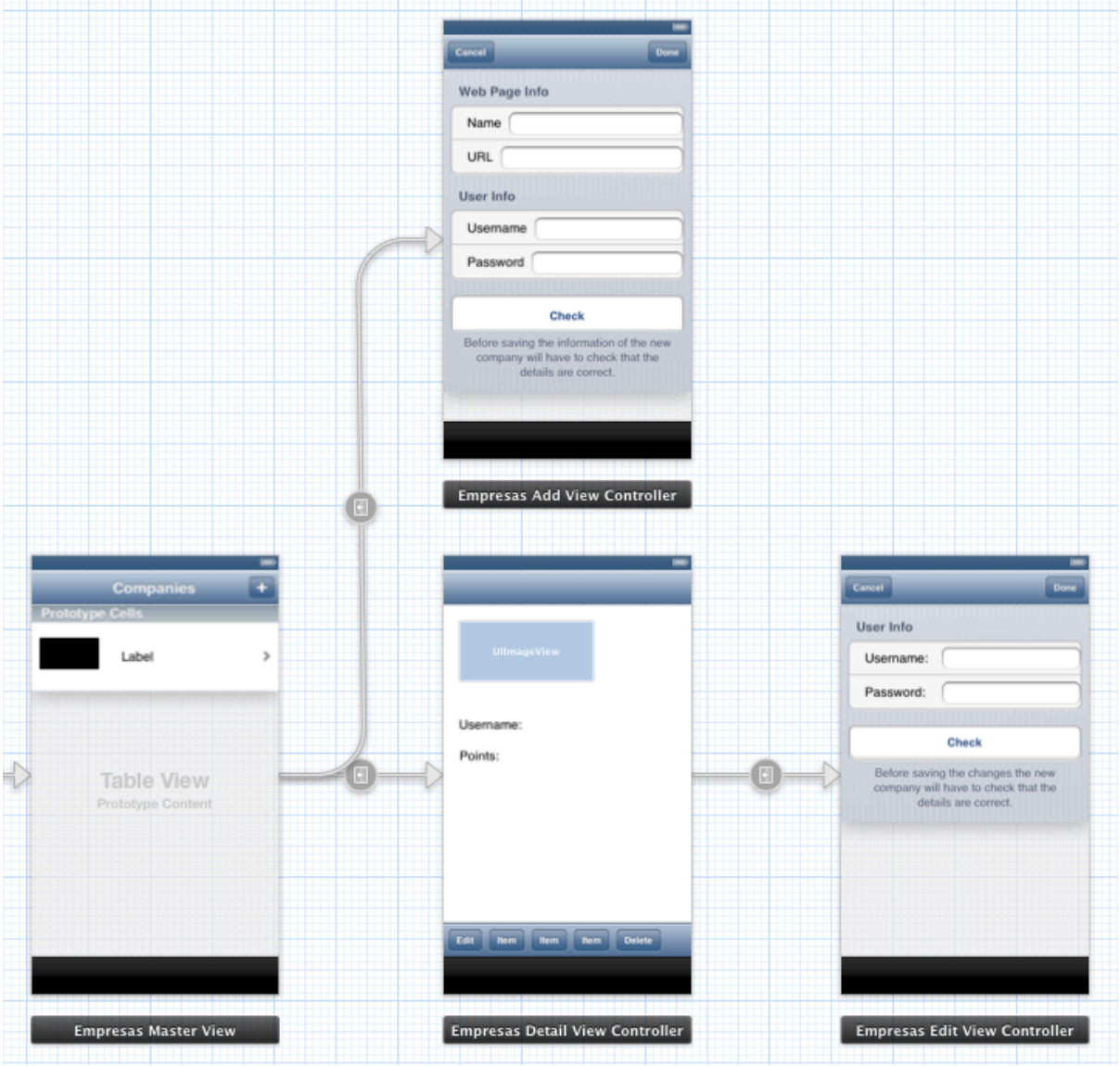

Figura 2. Diagrama de vistas 2

3.2.1.1 Vista inicial

La vista inicial es la primera vista que se muestra al usuario. La controla la clase *EmpresasRootViewController*. Muestra en pantalla una ventana en la que el usuario puede introducir un nombre de usuario y una contraseña para registrarse en la aplicación en el caso de ser la primera vez que entra en ella (véase Fig. 3).

Si ya se ha registrado, se muestra una ventana el nombre de usuario registrado en la que tiene que introducir su contraseña para poder acceder a la aplicación (véase Fig. 4).

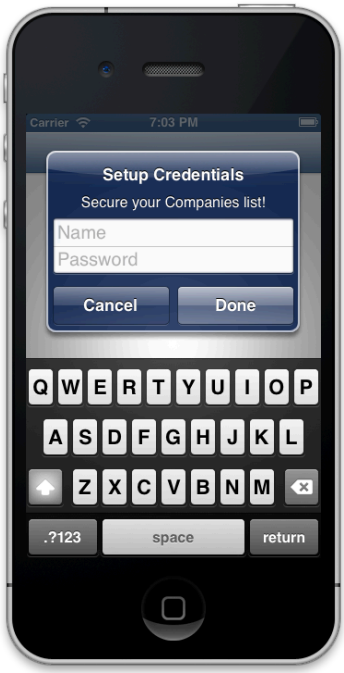

Figura 3. Vista inicial. Registro

Figura 4. Vista inicial. Autenticación

3.2.1.2 Lista de empresas

La lista de empresas se muestran en la siguiente vista, después de que el usuario se haya autenticado (véase Fig. 5). El controlador *EmpresasMasterViewController* hereda de la clase *UITableViewController* y muestra a través de su vista la lista de empresas registradas en la aplicación hasta el momento. Al pulsar sobre cada una de las empresas se pasa a la vista que muestra la información en detalle de la empresa seleccionada. También se puede pulsar sobre el botón + para registrar una nueva empresa.

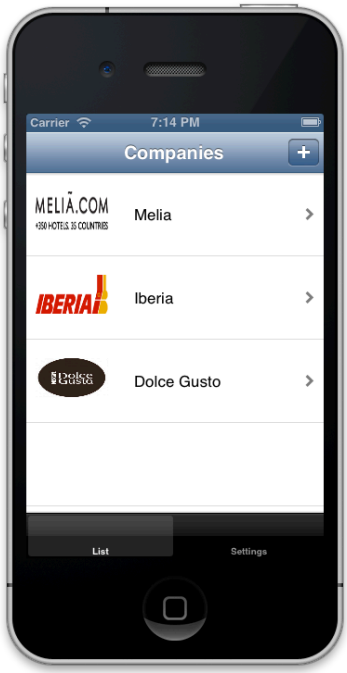

Figura 5. Vista lista de empresas

3.2.1.3 Detalles de empresa

Tras haber seleccionado una empresa se muestra su información a través de la vista gestionada por la clase *EmpresasDetailViewController*. Este controlador hereda de la clase *UIViewController*. Se puede observar como la barra de navegación indica la situación del usuario en la jerarquía de vistas. También contiene un botón para volver a la lista de empresas (véase Fig. 6).

Para esta vista se ha añadido una barra de herramientas con varios botones y se han implementado dos de ellos parar permitir al usuario borrar la empresa registrada o editarla.

Figura 6. Vista en detalle

3.2.1.4 Agregar nueva empresa

Muestra un formulario que permite introducir la información necesaria para registrar una nueva empresa (véase Fig. 7). El controlador de la vista, EmpresasAddViewController, no permite registrar la empresa hasta que no se comprueba que los datos introducidos son correctos y permiten al usuario autenticarse en el servidor.

Para comprobar los datos introducidos hay que pulsar el boto *Check,* una vez que se han verificado se habilita el botón *Done,* en la barra de navegación, para registrar la empresa.

En la barra de navegación también se ha añadido un botón para cancelar la operación y volver a la vista que muestra la lista de empresas.

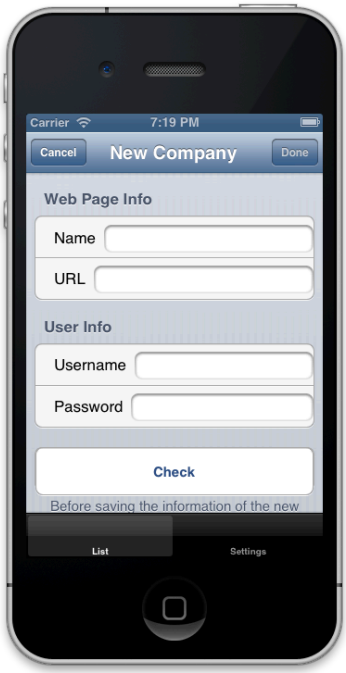

Figura 7. Vista registro de nueva empresa

3.2.1.5 Editar empresa

Esta vista pertenece a la clase *EmpresasDetailViewController*, se puede acceder a ella desde la clase *EmpresasDetailViewController*. Muestra un formulario para editar el nombre de usuario y la contraseña para la autenticación.

Al igual que en la vista anterior hay que verificar los datos. También se han añadido los botones *Done* y *Cancel* en la barra de navegación (véase Fig. 8).

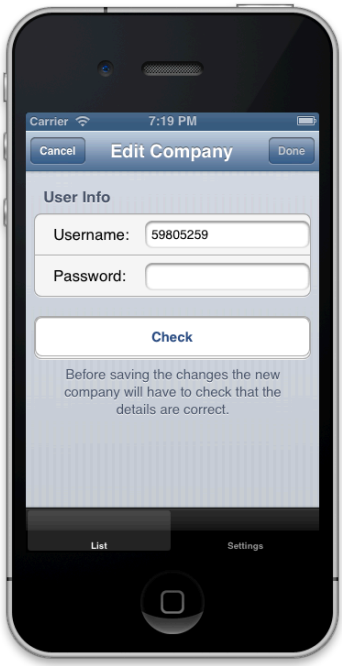

Figura 8. Vista edición de empresa

3.2.1.6 Ajustes

Hasta ahora se han visto las vistas pertenecientes a la pestaña *List* del *TabBarController*. Esta es la primera de las vistas de la pestaña *Settings*.

Muestra una lista con las diferentes acciones relacionadas con la configuración de la aplicación que se pueden realizar. Se ha añadido un solo elemento que permite cambiar el nombre de usuario y la contraseña para acceder a la aplicación (véase Fig. 9).

A esta vista se le ha asignado el controlador *EmpresasAjustesViewController* pero no se ha implementado ningún comportamiento personalizado.

Figura 9. Vista lista de ajustes

3.2.1.7 Cambiar contraseña

Pulsando sobre el elemento añadido a la lista de ajustes se accede a la vista del controlador *EmpresasAjustesPasswordViewController*. Muestra un formulario para cambiar el nombre de usuario y la contraseña de la aplicación (véase Fig. 10).

En la barra de navegación están los botones necesarios para aceptar el cambio o volver a la lista de ajustes.

Figura 10. Vista cambio de contraseña

3.3 Implementación

3.3.1 Puesta en marcha

Para desarrollar aplicaciones para iOS es necesario un ordenador Macintosh y las herramientas de Xcode junto con las librerías SDK [CFG10].

Xcode es el entorno de desarrollo integrado de Apple (IDE). Xcode ofrece todo lo necesario para crear aplicaciones para iOS. Incluye un editor de código fuente, un editor de interfaz gráfica de usuario, y muchas otras características. Utiliza una sola ventana, llamada la ventana de trabajo, que presenta la mayor parte de las herramientas que se necesitan para el desarrollo. Dentro de esta ventana se realiza la escritura de código la depuración y el diseño de la interfaz de usuario.

Xcode incluye kits de desarrollo de software (SDK) que le permiten crear aplicaciones que se ejecutan en versiones específicas de iOS, incluyendo versiones diferentes en la que se está desarrollando. Esta tecnología le permite crear un único archivo binario que aprovecha las nuevas características cuando se ejecuta en un sistema que las soporta, y se degrada, manteniendo la compatibilidad, cuando se ejecuta en un sistema antiguo.

Para instalar Xcode hay que realizar los siguientes pasos:

1. La aplicación App Store se instala con la versión Mac OS X 10.7 y posteriores. Por lo que si se tiene una versión anterior de Mac OS X, es necesario actualizarla.

2. Hay que descargar la versión más reciente de Xcode. Se abre la aplicación App Store en el Mac, se busca la aplicación Xcode, y se hace clic en libre para descargarla.

3.3.1.1 Xcode

Xcode [CFG10] es una suite de herramientas de desarrollo de Apple que proporcionan soporte para la gestión de proyectos , edición de código, compilación de ejecutables, depuración del código, ajuste de rendimiento y mucho más. En el centro de esta suite está la aplicación Xcode en sí misma, que proporciona el entorno de desarrollo básico.

La aplicación Xcode es un entorno de desarrollo integrado (IDE), que proporciona todas las herramientas necesarias para crear y gestionar proyectos de iOS y los archivos de código fuente. Facilita el desarrollo de la interfaz de usuario, permite crear el código en un archivo ejecutable, ejecutar y depurar el código, ya sea en simulador de iOS o en un dispositivo. Xcode incorpora una serie de características para hacer más fácil el desarrollo de aplicaciones de iOS, incluyendo las siguientes:

- Un sistema de gestión de proyectos para la definición de los productos de software.
- Un entorno de edición de código que incluye características tales como coloreado de sintaxis y completado de código.
- Un editor integrado para la creación de stroyboards y archivos nib.
- Un visor de documentos avanzado para la visualización y búsqueda de documentación de Apple.
- Un inspector contextual para ver la información acerca del código seleccionado.
- Un avanzado sistema de compilación con comprobación de dependencias y construcción de reglas de evaluación.
- Soporte para C, C + + y Objective-C.
- Compiladores GCC con soporte para C, C + +, Objective-C, Objective-C + + y otros lenguajes.
- Un analizador estático para validar el comportamiento de la aplicación y la identificación de

problemas potenciales.
- Depuración integrada a nivel de fuente usando GDB.
- Apoyo a la gestión de código fuente integrado.
- Apoyo a la gestión de dispositivos de desarrollo en iOS.

Para crear una nueva aplicación de iOS, se inicia mediante la creación de un nuevo proyecto en Xcode. Un proyecto gestiona toda la información relacionada con su solicitud, incluyendo los archivos de código fuente, construir valores y las normas necesarias para poner todas las piezas juntas. El corazón de cada proyecto Xcode es la ventana del proyecto, que se muestra en la Figura A-1. Esta ventana proporciona acceso rápido a todos los elementos clave de su aplicación. En los Grupos y la lista de archivos, administrar los archivos del proyecto, incluidos los archivos de origen y construir objetivos que se crean a partir de los archivos de origen. En la barra de herramientas, puede acceder a las herramientas y los comandos utilizados frecuentemente. A continuación, puede configurar el área de trabajo para mostrar los paneles necesarios para la edición, navegación de su contenido del proyecto, depuración, y obtener información adicional acerca de los elementos.

3.3.1.2 Simulador de iOS

El simulador de iOS [CFG10] permite probar rápidamente prototipos y versiones de su aplicación durante el proceso de desarrollo. Instalado como parte de las herramientas Xcode y junto con el SDK de iOS, el simulador de iOS se ejecuta en el Mac y se comporta como una aplicación para Mac estándar, mientras que simula el entorno de un iPhone o un iPad. El simulador es como una herramienta de pruebas preliminares que se usa antes de probar la aplicación en un dispositivo real.

El simulador de iOS permite simular varios dispositivos iOS y varias versiones del sistema operativo iOS. Cada versión de software simulado se considera como su propio entorno de simulación, independiente de las otras versiones, con sus propios valores y archivos. Estos ajustes y archivos existen para todos los dispositivos a probar dentro de un entorno de simulación.

3.3.2 Especificación de las clases

El proyecto está compuesto por un conjunto de clases y protocolos. Para una mejor comprensión se van a diferenciar las clases: las que actúan como controladores de vista y las que no se han asignado a ninguna de las vistas de la aplicación pero se encargan de un conjunto de tareas relacionadas. Se van a comentar los métodos más importantes de cada clase.

3.3.2.1 Controladores de vista

Cada una de estas clases se ha asignado a una de las vistas comentadas anteriormente, actuando como su controlador (véase de la Fig. 11 a la Fig. l6).

Clase EmpresasRootViewController	
Esta clase pertenece a la vista inicial de la aplicación en la que se le muestra al usuario una ventana en la que tiene que autenticarse para poder acceder a la aplicación.	
viewDidLoad	Se llama al método después de que se haya cargado en memoria la vista del controlador. Se usa para inicializar variables o llamar a otros métodos.
viewDidAppear:	Notifica al controlador de la vista que su vista se ha añadido a la jerarquía de vistas. Se puede sobrescribir este método para personalizar la presentación de la vista.
presentAlertViewForPassword	Muestra al usuario una ventana en la que autenticarse para acceder a la aplicación o para registrarse si es la primera vez que entra.
alertView: didDismissWithButtonIndex:	Se llama al método cuando el usuario acepta o cancela la autenticación.
credentialsValidated	Indica si el usuario se ha registrado.
textFieldDidEndEditing:	Es un método delegado que permite examinar el texto introducido en los campos de texto. Se usa para comprobar si la clave introducida es correcta y para registrar al usuario.
prepareForSegue: sender:	Es un método delegado al que se llama cuando una transición entre vistas está a punto de realizarse. Se sobrescribe para identificar hacia que vista se produce la transición y para pasar datos la otra vista.

Figura 11. Clase EmpresasRootViewController

Clase EmpresasMasterViewController	
Hereda de la clase *UITableViewController*, muestra la lista de las empresas registradas. También se encarga de guardar, editar y eliminar los elementos de la lista.	
viewDidLoad	Explicado anteriormente, en esta clase se usa para inicializar variables y para realizar una búsqueda de las empresas registradas.
numberOfSectionsInTableView:	Es un método heredado de la clase *UITableViewController* que devuelve el número de secciones de la tabla.
tableView: numberOfRowsInSection:	Es un método heredado de la clase *UITableViewController* que devuelve el número de filas en una sección.
tableView: cellForRowAtIndexPath:	Es el método encargado de insertar cada celda en la tabla en una posición determinada. También se hereda de la clase *UITableViewController*.
tableView: canEditRowAtIndexPath:	A través de este método se indica si se quiere que las celdas se puedan editar.
modificaEmpresaP: withIndexPath: usuario: clave:	Se encarga de modificar el usuario y la clave de autenticación de una empresa. Está implementado en esta clase pero declarado en el protocolo *EmpresasProtocol*.
eliminaEmpresaP: withRow:	Se encarga de eliminar una empresa registrada. Está implementado en esta clase pero declarado en el protocolo *EmpresasProtocol*.
agregaEmpresaP: withEmpresa: password: index:	Se encarga de registrar una nueva empresa. Está implementado en esta clase pero declarado en el protocolo *EmpresasProtocol*.
prepareForSegue: sender:	Mencionado anteriormente. En esta clase se distinguen dos transiciones: hacia la clase *EmpresasAddViewController* para añadir una nueva empresa y hacia la clase *EmpresasDetailViewController* para mostrar la información de la empresa seleccionada.

Figura 12. Clase EmpresasMasterViewController

Clase EmpresasDetailViewController	
Hereda de la clase *UIViewController*, muestra información de la empresa seleccionada en la lista de empresas.	
viewDidLoad	Explicado anteriormente, en este caso se configura la vista y se inicia la conexión para autenticarse en el servidor y actualizar los datos de la empresa seleccionada.
resultadoConexionPost: result:	Una vez finalizada la conexión con el servidor se obtiene el resultado a través de este método delegado. Obtenida la información actualiza los datos de la empresa o comunica al usuario si ha habido algún error.
editSaveButtonTapped: withIndexPath: usuario: clave:	Es un método delegado que actúa de puente entre la clase *EmpresasEditViewController* y la clase *EmpresasMasterViewController* para modificar una empresa.
editCancelButtonTapped	Es un método delegado cambia a la vista anterior.
eliminaEmpresa:	Es la acción del botón *Delete* de la vista de este controlador. Llama al método *eliminaEmpresaP: withRow:* para borrar la empresa actual.
prepareForSegue: sender:	En este caso también se produce una transición hacia la vista *EmpresasEditViewController*.

Figura 13. Clase EmpresasDetailViewController

Clase EmpresasAddViewController	
Muestra un formulario con los campos necesarios para registrar una nueva empresa. Hereda de la clase *UITableViewController*.	
viewDidLoad	Inicializa varias variables.
resultadoConexionPost: result:	Devuelve el resultado de la consulta POST al servidor. Si ha sido exitoso se activa el botón *Done*.
resultadoConexionGet: result:	Devuelve el resultado de la consulta GET al servidor. Si se ha descargado la imagen del logo de la empresa se registra la nueva empresa.
done:	Es la acción del botón *Done*. Está activo cuando se han comprobado que los datos introducidos por el usuario son correctos y se ha podido realizar la autenticación. En este método se realiza la consulta GET para obtener la imagen del logo de la empresa.
cancel:	Es la acción del botón *Cancel.* Vuelve a la vista anterior.
check:	Es la acción del botón *Check,* envía una consulta POST para comprobar que los datos introducidos por el usuario son correctos.

Figura 14. ClaseAddViewController

Clase EmpresasEditViewController	
Muestra un formulario con los campos necesarios editar el nombre de usuario y la contraseña de una empresa ya registrada. Hereda de la clase *UITableViewController*.	
viewDidLoad	Inicializa varias variables.
resultadoConexionPost: result:	Devuelve el resultado de la consulta POST al servidor. Si ha sido exitoso se activa el botón *Done*.
done:	Es la acción del botón *Done*. Está activo cuando se han comprobado que los datos introducidos por el usuario son correctos y se ha podido realizar la autenticación. En este método se llama al método delegado *editSaveButtonTapped: withIndexPath: usuario: clave:* para que se guarden los cambios realizados.
cancel:	Es la acción del botón *Cancel.* Vuelve a la vista anterior.
check:	Es la acción del botón *Check,* envía una consulta POST para comprobar que los datos introducidos por el usuario son correctos.

Figura 15. Clase EmpresasEditViewController

Clase EmpresasAjustesPasswordViewController	
Muestra un formulario con los campos necesarios editar el nombre de usuario y la contraseña de la aplicación. Hereda de la clase *UITableViewController*.	
viewDidLoad	Inicializa varias variables y obtiene el contexto de la clase delegada del proyecto.
modificaEmpresaWithIndexPath: usuario: clave:	Modifica el nombre de usuario y la contraseña de la empresa.
eliminaEmpresaWithRow:	Elimina la empresa.
agregaEmpresaWithEmpresa: password: index:	Añade una nueva empresa.
done:	Registra al nuevo usuario en la aplicación y vuelve a codificar las contraseñas guardadas de todas las empresas ya que usan como llave la clave del usuario.

Figura 16. Clase EmpresasAjustesPasswordViewController

3.3.2.2 Otras Clases

Las clases que se muestran a continuación no actúan como controladores de vista. Las tareas que implementan estas clases se realizan en diferentes puntos de la aplicación. Están relacionadas con la comunicación con los servidores, la autenticación y el almacenamiento seguro de claves (véase Fig. 17 y Fig. 18).

Clase EmpresasConnection	
Realiza todas las tareas relacionadas con la comunicación con el servidor mediante peticiones HTTP. Abre la conexión, construye las peticiones, recibe la respuesta del servidor y extrae la información de interés. Estos métodos se verán más en detalle en los apartados para realizar consultas HTTP y autenticación.	
initWithEmpresa: password: delegate:	Inicializa un objeto EmpresasConnection con un objeto de tipo Empresas, un NSString con la clave de la empresa y la clase desde la que se crea el objeto.
startReceiveGet	Inicia la conexión para realizar una consulta de tipo GET.
startReceivePost	Inicia la conexión para realizar una consulta de tipo POST.
pathForTemporaryFileWithPrefix:	Devuelve la ruta del directorio temporal para el usuario actual.
connection: canAuthenticateAgainstProtectionSpace:	Método delegado que se ejecuta cuando el servidor requiere de algún tipo de autenticación.
connection: didReceiveAuthenticationChallenge:	Método delegado para tratar el desafío de autenticación.
connection: didReceiveResponse:	Método delegado que examina el tipo de respuesta del servidor.
connection: didReceiveData:	Método delegado para recibir los datos de la respuesta del servidor.
connectionDidFinishLoading:	Cuando la conexión finaliza con éxito se ejecuta este método delegado.
connection: didFailWithError:	Método delegado que se ejecuta si se produce algún error durante la conexión.
stopReceiveWithStatus:	En este método se cierran todas las conexiones y los streams.
receiveDidStopWithStatus:	Se tratan los datos recibidos y se pasan a otras clases.
findInHtmlOtro;	Cuando el host es desconocido la aplicación entra en este método registrando el error.
findInHtmlWith: desplazamiento: primerCaracter:	Busca en el código html recibido la información de interés.

Figura 17. Clase EmpresasConnection

Clase EmpresasKeychainWrapper	
Realiza todas las tareas relacionadas con el llavero. Guarda las claves cifradas y las obtiene del llavero cuando se soliciten. Estos métodos se verán con más detalle en el apartado de persistencia de datos.	
searchKeychainCopyMatchingIdentifier:	Devuelve los datos de clave guardada en el llavero con un identificador determinado.
keychainStringFromMatchingIdentifier:	Llama a la función anterior y devuelve la clave en formato *NSString*.
compareKeychainValueForMatchingPIN:	Compara la clave introducida por el usuario con la clave cifrada guardada en el llavero.
createKeychainValue: forIdentifier:	Guarda una clave en el llavero en formato NSString con un identificador.
createKeychainData: forIdentifier:	Guarda una clave en el llavero en formato NSData con un identificador.
updateKeychainValue: forIdentifier:	Actualiza una clave en el llavero en formato NSString para un identificador existente.
updateKeychainData: forIdentifier:	Actualiza una clave en el llavero en formato NSData para un identificador existente.
deleteItemFromKeychainWithIdentifier:	Elimina el elemento del llavero con un identificador determinado.
securedSHA256DigestHashForPIN:	Devuelve un objeto de tipo *NSString* con la clave encriptada mediante el algoritmo SHA.
computeSHA256DigestForString:	El método anterior le pasa la cadena de caracteres a la que finalmente aplica el algoritmo de SHA para cifrar la clave.
AES256EncryptWithKey: clave:	Cifra la clave aplicando el algoritmo AES.
AES256DecryptWithKey: clave:	Desencripta la clave encriptada mediante el algoritmo AES.
guardarClave: withNombreEmpresa: key:	Guarda en el llavero la clave de una empresa encriptada mediante el algoritmo AES, usando la contraseña del usuario como llave.
obtenerClaveWithNombreEmpresa: key:	Obtiene una clave guardada en el llavero y encriptada mediante el algoritmo AES, usando la contraseña del usuario como llave.

Figura 18. Clase EmpresasKeychainWrapper

3.3.2.3 Protocolos

Los protocolos se usan para declarar métodos delegados. Estos métodos ya se han explicado porque se implementan en las clases anteriores (véase de la Fig. 19 a la Fig. 21). Tienen como finalidad pasar objetos entre las diferentes vistas o indicar a otra clase que un proceso ha terminado. Esto se explicará con más detalle en el apartado de métodos delegados.

Protocolo EmpresasProtocol
modificaEmpresaP: withIndexPath: usuario: clave:
eliminaEmpresaP: withRow:
agregaEmpresaP: withEmpresa: password: index:

Figura 19. Protocolo EmpresasProtocol

Protocolo EmpresasEditProtocol
editSaveButtonTapped: withIndexPath: usuario: clave:
editCancelButtonTapped

Figura 20. Protocolo EmpresasEditProtocol

Protocolo EmpresasConnectionProtocol
resultadoConexionPost: result:
resultadoConexionGet: result:

Figura 21. Protocolo EmpresasConnectionProtocol

3.3.3 Codificación

3.3.3.1 Persistencia de datos

Para guardar los datos de forma persistente se usa el framework Core Data
[CDPG12]. Proporciona soluciones automatizadas para tareas comunes asociadas con la persistencia de los datos y la gestión gráfica de objetos.

Se caracteriza por mantener la relación entre los objetos, reducir la sobrecarga de memoria, intercambio de datos y la validación automática del valor de las propiedades. Permite agrupar, filtrar y organizar los datos en memoria y en la interfaz del usuario. También soporta búsquedas complejas.

La mayor parte de esta funcionalidad la proporciona de forma automática a través de un objeto conocido como “managed object context”o contexto.

Los modelos de objetos en el framework Core Data son conocidos como “managed objects”o objetos gestionados.

Entre los contextos y los objetos guardados de forma persistente hay un “Persistent Store Coordinator”o Coordinador de Almacenamiento Persistente.

En la Fig. 22 se puede observar claramente la relación entre los diferentes elementos:

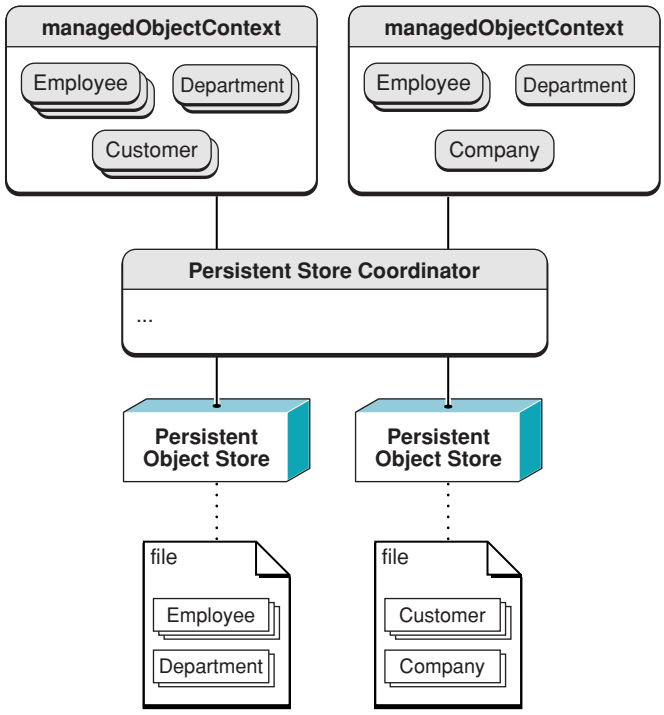

Figura 22. Elementos de la pila Core Data

3.3.3.1.1 Creación de la pila Core Data

Los diferentes elementos se crean en la clase delegada del proyecto (*EmpresasAppDelegate*) de manera que sea accesible en las demás clases.

```
@property (readonly, strong, nonatomic) NSManagedObjectContext
*managedObjectContext;
@property (readonly, strong, nonatomic) NSManagedObjectModel *managedObjectModel;
@property (readonly, strong, nonatomic) NSPersistentStoreCoordinator
*persistentStoreCoordinator;


- (void)saveContext;
- (NSURL *)applicationDocumentsDirectory;
- (void)setMasterEmpresasList:(NSMutableArray *)masterEmpresasList;
```

Figura 23. Código EmpresasAppDelegate.h

En la cabecera, *EmpresasAppDelegate.h*, se definen los objetos o propiedades de la clase y se declaran los métodos (véase Fig. 23). Estos se implementan en el archivo *EmpresasAppDelegate.m*. En la Fig. 24 se muestra parte del código de la clase delegada.

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // Override point for customization after application launch.
    NSManagedObjectContext *context = [self managedObjectContext];
    if (!context) {
        // Handle the error.
    }
    return YES;
}
// Returns the managed object context for the application.
// If the context doesn't already exist, it is created and bound to the persistent
store coordinator for the application.

- (NSManagedObjectContext *)managedObjectContext
{
    if (_managedObjectContext != nil) {
        return _managedObjectContext;
    }
    NSPersistentStoreCoordinator *coordinator = [self persistentStoreCoordinator];
    if (coordinator != nil) {
        _managedObjectContext = [[NSManagedObjectContext alloc] init];
        [_managedObjectContext setPersistentStoreCoordinator:coordinator];
    }
    return _managedObjectContext;
}
// Returns the managed object model for the application.
// If the model doesn't already exist, it is created from the application's model.
- (NSManagedObjectModel *)managedObjectModel
{
    if (_managedObjectModel != nil) {
        return _managedObjectModel;
    }
    NSURL *modelURL = [[NSBundle mainBundle] URLForResource:@"Empresas"
withExtension:@"momd"];
    _managedObjectModel = [[NSManagedObjectModel alloc]
initWithContentsOfURL:modelURL];
    return _managedObjectModel;
}
// Returns the persistent store coordinator for the application.
// If the coordinator doesn't already exist, it is created and the application's
store added to it.
- (NSPersistentStoreCoordinator *)persistentStoreCoordinator
{
    if (_persistentStoreCoordinator != nil) {
        return _persistentStoreCoordinator;
    }
    NSURL *storeURL = [[self applicationDocumentsDirectory]
URLByAppendingPathComponent:@"Empresas.sqlite"];

    NSError *error = nil;
    _persistentStoreCoordinator = [[NSPersistentStoreCoordinator alloc]
initWithManagedObjectModel:[self managedObjectModel]];
    if (![_persistentStoreCoordinator addPersistentStoreWithType:NSSQLiteStoreType
configuration:nil URL:storeURL options:nil error:&error]) {
NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
        abort();
    }
    return _persistentStoreCoordinator;
}
```

Figura 24. Código EmpresasAppDelegate.m

El método *Application:didFinishLaunchingWithOptions:* comunica al delegado que el proceso de puesta en marcha está casi terminado y la aplicación está casi listo para funcionar. Se utiliza este método para completar la inicialización de la aplicación y haga los ajustes finales. En este caso se usa para inicializar los elementos de la pila Core Data.

El modelo se inicializa mediante el método *managedObjectModel*, lo hace a través de la función *initWithContentsOfURL:* a la que hay que pasarle la URL o ruta del archivo del modelo. La ruta se obtiene con el método *URLForResource:withExtension:* que devuelve la URL del archivo para el recurso identificado por el nombre y la extensión del archivo.

El Coordinador de Almacenamiento Persistente se inicializa con el método *initWithManagedObjectModel:* pasándole el modelo de objeto como parámetro.

Por último, se inicializa el contexto. Una vez inicializado se le asigna el un Coordinador de Almacenamiento Persistente con el método *setPersistentStoreCoordinator:coordinator:*, pasándolo como parámetro.

En la Fig. 25 se muestran las líneas de código para poder acceder al contexto desde cualquier clase. Se pueden implementar, por ejemplo, en el método *viewDidLoad*.

```
self.appDelegate = (EmpresasAppDelegate *)[[UIApplication
sharedApplication]delegate];

self.managedObjectContext  = [self.appDelegate managedObjectContext];
```

Figura 25. Código acceso al contexto

Siendo *appDelegate* una propiedad de la clase desde la que se quiere acceder al contexto. De esta forma se accede a la clase delegada del proyecto, tanto a las propiedades como a los métodos.

3.3.3.1.2 Definir el modelo de objeto gestionado con Core Data Model Editor

Es necesario especificar la estructura de los datos, es decir, hay que definir los atributos de los objetos gestionados o entidades que se van a guardar de forma persistente. Para ello se puede usar Core Data Editor [CDME12]. Es una utilidad de Xcode que permite crear de forma gráfica el modelo de los datos que se van a guardar de forma persistente. Para ello se tienen que seguir los siguientes pasos (véase Fig. 26):

- En Xcode seleccionar File → New → File...
- En la nueva ventana selecciona Core Data y a la derecha el icono Data Model tal y como se muestra en la siguiente figura.
- Tras pulsar next, salvar el modelo con el nombre deseado.

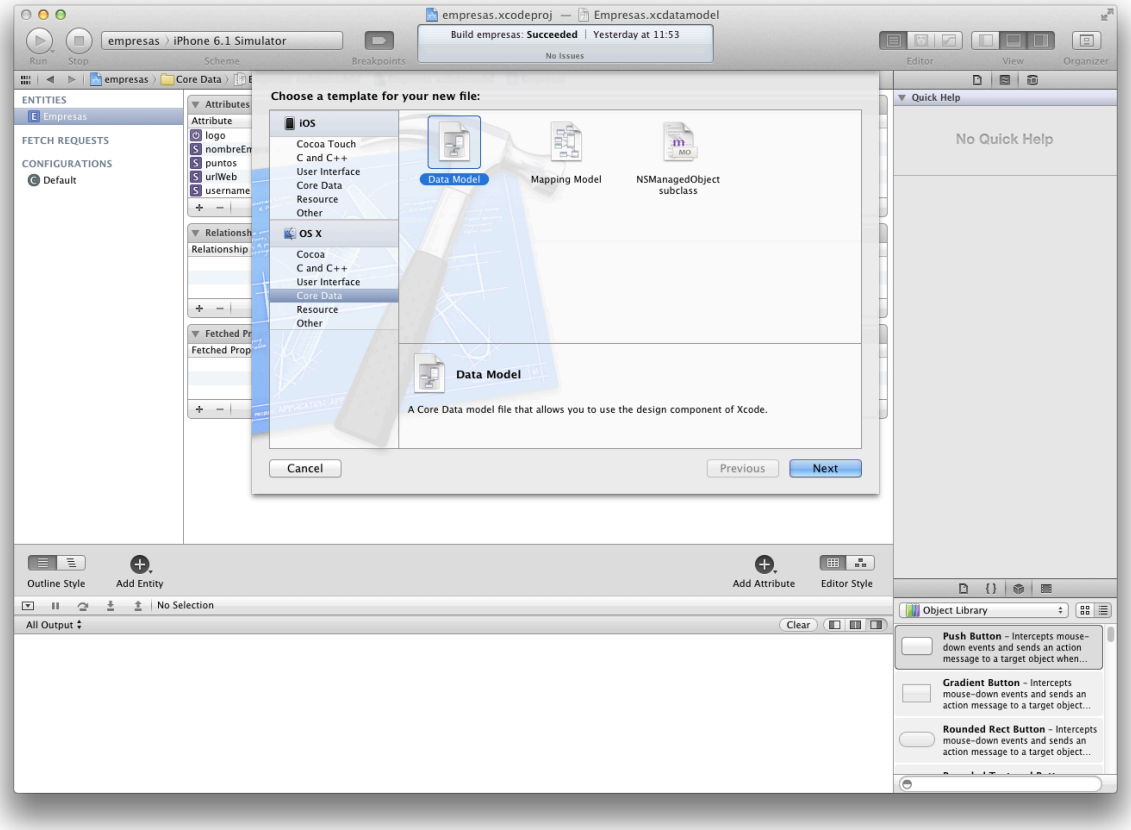

Figura 26. Model Editor. Nuevo modelo

Una vez creado el modelo se añade la entidad y se crean sus atributos. Para crear la entidad (véase Fig. 27):

- Pulsar Add Entity en la parte inferior y especificar el nombre de la entidad.
- Seleccionada la entidad en el apartado de atributos se pueden añadir los atributos deseados pulsando sobre el "+".

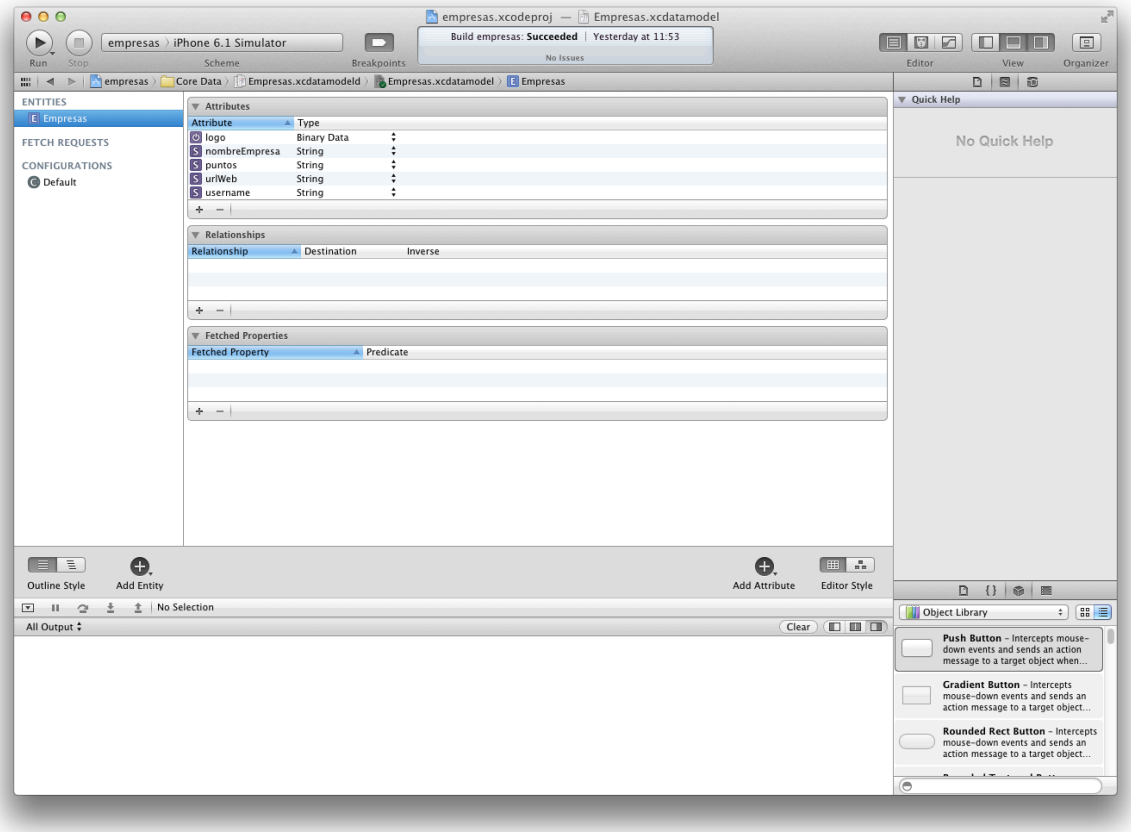

Figura 27. Model Editor. Nueva entidad

Cuando se ha especificado el modelo hay que crear el código fuente de la entidad creada.

- Se selecciona la entidad o entidades de las que se quiere crear una subclase NSManagedObject.
- Seleccionar Editor → Create NSManagedObject subclass.
- En el nuevo diálogo seleccionar la carpeta donde se guardará el nuevo archivo.
- Pulsar en créate.

En este proyecto se ha creado el objeto Empresas, definido por dos archivos: *Empresas.h* y *Empresas.m* donde se pueden observar sus propiedades. El archivo *Empresas.h* se muestra en la Fig. 28.

```
#import <Foundation/Foundation.h>
#import <CoreData/CoreData.h>


@interface Empresas : NSManagedObject
@property (nonatomic, retain) NSString * nombreEmpresa;
@property (nonatomic, retain) NSString * puntos;
@property (nonatomic, retain) NSString * urlWeb;
@property (nonatomic, retain) NSString * username;
@property (nonatomic, retain) NSData * logo;
@end
```

Figura 28. Código Empresas.h

3.3.3.1.3 Realizar peticiones de búsqueda

Las peticiones de búsqueda las hace el controlador de vista *EmpresasMasterViewController*, ya que su vista es la que muestra al usuario la lista de empresas registradas, y se obtienen de los datos almacenados.

Para obtener los datos requeridos hay que realizar una consulta de búsqueda. En la consulta se tiene que especificar la entidad y el contexto al que pertenece. También hay que elegir uno de los atributos del objeto gestionado que se usa para ordenar el resultado de la búsqueda, de forma ascendente o descendente. El código de la Fig. 29 pertenece al método *viewDidLoad*, que es llamado después de que la vista del controlador se haya cargado en memoria.

```
NSFetchRequest *request = [[NSFetchRequest alloc] init];
    NSEntityDescription *entity = [NSEntityDescription entityForName:@"Empresas"
inManagedObjectContext:managedObjectContext];
    [request setEntity:entity];

    NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc]
                                        initWithKey:@"nombreEmpresa"
ascending:NO];
    NSArray *sortDescriptors = [[NSArray alloc] initWithObjects:sortDescriptor,
nil];
    [request setSortDescriptors:sortDescriptors];

    NSError *error = nil;
    NSMutableArray *mutableFetchResults = [[managedObjectContext
executeFetchRequest:request error:&error] mutableCopy];
    if (mutableFetchResults == nil) {
        // Handle the error.
    }
    [self setMasterEmpresasList:mutableFetchResults];
```

Figura 29. Código consulta de búsqueda

Se inicializa la petición de búsqueda y se crea un objeto de tipo *NSEntityDescription* mediante el método *entityForName:inManagedObjectContext:*, que se le pasa como parámetros la entidad deseada del modelo de objetos y el contexto.

El objeto *NSSortDescriptor* se usa para escoger la propiedad y el orden que seguirá el resultado de la búsqueda. En este caso se elige el atributo *nombreEmpresa* en orden descendente. Este objeto se añade a la petición de búsqueda mediante el método *setSortDescriptors:*.

Finalmente se realiza la búsqueda enviando el mensaje *executeFetchRequest:error:* al contexto. Como resultado se obtiene un array con los objetos requeridos pertenecientes a la entidad que se ha especificado. Este resultado se asigna a la propiedad *masterEmpresasList*, que es el array que usa la vista para mostrar la lista de empresas al usuario.

3.3.3.1.4 Guardar datos en la pila

Conviene hacer un pequeño diagrama para entender la relación entre las diferentes clases que intervienen (véase Fig. 30). Tomando la siguiente figura como referencia se va a explicar las tareas que intervienen para guardar una nueva empresa de forma persistente.

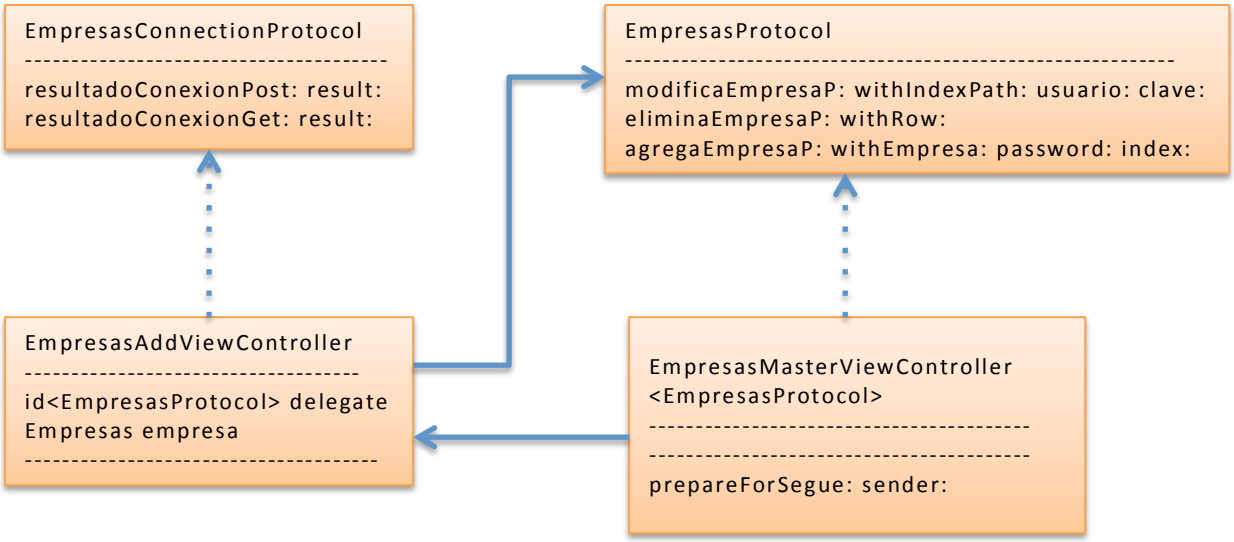

Figura 30. Relación entre clases. Registro nueva empresa

La clase *EmpresasMasterViewController* implementa los métodos definidos en el protocolo *EmpresasProtocol* a la vez crea un objeto de la clase *EmpresasAddViewController*. Mientras que esta implementa los métodos del protocolo *EmpresasConnectionProtocol* y tiene una propiedad definida conforme al protocolo *EmpresasProtocol*. Estas relaciones se establecen básicamente para pasar información entre clases e indicar a otras clases que han finalizado ciertas operaciones haciendo uso de métodos delegados.

Si el usuario quiere registrar una nueva empresa tiene que pulsar sobre el botón + en la vista del controlador *EmpresasMasterViewController*. Al realizar esta acción se realiza una transición al controlador *EmpresasAddViewController*. Justo antes de realizar la transición se llama al método *prepareForSegue*. El código de esta función para la transición se muestra en la Fig. 31.

```
if ([[segue identifier] isEqualToString:@"addEmpresa"]) {
        // Create and configure a new instance of the Empresas entity.
        Empresas *nuevaEmpresa = (Empresas *)[NSEntityDescription
insertNewObjectForEntityForName:@"Empresas"
inManagedObjectContext:managedObjectContext];
        EmpresasAddViewController *addController = [segue
destinationViewController];
        addController.empresa = nuevaEmpresa;
        addController.delegate = self;
    }
```

Figura 31. Código transición hacia EmpresasAddViewController

Para guardar los datos en la pila Core Data hay se crea una nueva instancia de la entidad, se realiza mediante el método *insertNewObjectForEntityForName:*. Para hacer la transición se crea un objeto de la clase *EmpresasAddViewController* y se le pasa la nueva instancia creada asignándola a su propiedad *empresa*. También asigna su propiedad *delegate* la clase actual.

En la clase *EmpresasAddViewController*, cuando se han verificado los datos de la empresa, en el método *resultadoConexionGet:result:*, implementado en esta clase, se llama al método *agregaEmpresaP:withEmpresa:password:index:*. Las nuevas empresas se guardan de forma

persistente este método, implementado en la clase *EmpresasMasterViewController*.

En la Fig. 32 se muestra parte del código de la función *agregaEmpresaP:withEmpresa:password:index:*. Los cambios realizados en el contexto no se guardarán de forma persistente hasta que no se guarde el contexto mediante la función *save*.

```
NSError *error = nil;
if (![managedObjectContext save:&error]) {
        NSLog(@"Error: %@",error);
}
```

Figura 32. Código guardar contexto

3.3.3.2 Realizar consultas HTTP

iOs proporciona varias APIs de propósito general para realizar consultas HTTP y HTTPS [INO13]. Con estas APIs se pueden descargar ficheros y realizar consultas simples de tipo GET o POST. En este proyecto se han usado las clases del framework *Foundation* para interactuar con URLs y poder comunicarse con servidores usando protocolos estándar de Internet.

El framework Foundation proporciona una rica colección de clases que incluyen soporte para trabajar con URLs [ULSP10], almacenamiento de cookies, autenticación y almacenamiento de credenciales. Concretamente la clase NSURL permite manipular URLs y los recursos a los que se refieren.

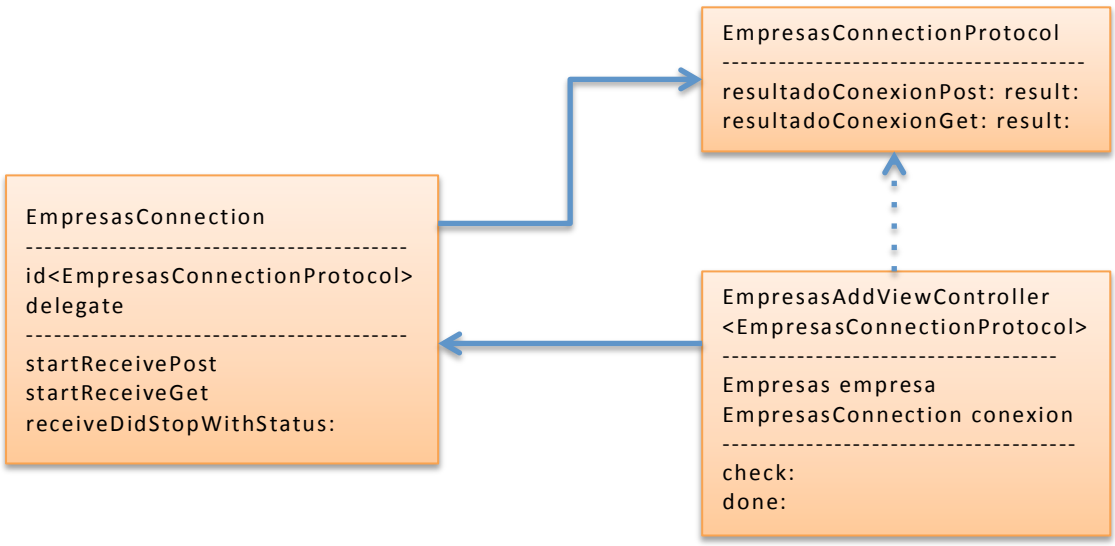

Figura 33. Relación entre clases. Consultas HTTP

En el proyecto, todas las tareas referentes a la conexión con los servidores las realiza la clase *EmpresasConnection* junto con el protocolo *EmpresasConnectionProtocol*. La relación entre esta clase y los controladores de vista se puede observar en la Fig. 33.

Se ha usado como ejemplo la clase *AddViewController* porque, aunque otros controladores de vista hacen uso de la clase *EmpresasConnection*, es la única que realiza los dos tipos de consultas: consultas de tipo GET y POST. En esta apartado se van a tratar las consultas de tipo GET, el otro tipo

de consultas se verán en el apartado de autenticación.

La clase *AddViewController* contiene una propiedad de tipo *EmpresasConnection* llamada *conexión.* Esta propiedad se inicializa en el método *check* mediante la función *initWithEmpresa:password:delegate:self*, tal y como muestra la Fig. 34. Hay que destacar que se le pasa como parámetro delegado la propia clase.

```
//Se crea la conexión
self.Connection = [[NSURLConnection alloc] initWithRequest:getRequest
delegate:self];
```

Figura 34. Código creación de la conexión

La clase *EmpresasConnection* tiene una propiedad conforme al protocolo *EmpresasConnectionProtocol* denominada *delegate.* En el método inicializador *initWithEmpresa:password:delegate:self*, se le asigna la clase *AddViewController* pasada como parámetro a la propiedad *delegate* (véase Fig. 35).

```
-(id)initWithEmpresa:(Empresas *)empresa password:(NSString *)password
delegate:(id)delegate {
    self = [super init];

    if (self) {
        self.empresa = empresa;
        self.password = password;
        self.delegate = delegate;
        self.get = false;
        self.post = false;
    }
    return self;
}
```

Figura 35. Código método constructor de la clase EmpresasConnection

Como se ha comentado anteriormente, una vez que se han verificado los datos introducidos por el usuario, se habilita el botón *Done* de la vista que envía un mensaje *startReceiveGet* al objeto *conexión* ya inicializado (véase Fig. 36).

```
- (IBAction)done:(id)sender {
    //Hasta que no se han comprobado los datos no se registra la empresa
    if (!self.post) {
        [self.conexion startReceiveGet];
    }
}
```

Figura 36. Código acción done de la clase EmpresasAddViewController

La clase *AddViewController* hace uso del protocolo *EmpresasConnectionProtocol*, es decir, tiene que implementar los métodos definidos en el protocolo. Por lo que, usando estos métodos, la clase *EmpresasConnection* puede comunicarle cuando ha finalizado la conexión con el servidor y el resultado obtenido.

```
}else if (self.get) {
        //Se ha solicitado el logo
        assert(self.fileStream == nil);
        self.datosImagen = [[NSData alloc]initWithContentsOfFile:self.filePath];
        if ([self.datosImagen length] == 0) {
            result = false;
            if (statusString == nil) {
                statusString = @"Failed to download logo";
            }
        }else {
            result = true;
            self.empresa.logo = [[NSData alloc]initWithData:self.datosImagen];
            if (statusString == nil) {
                statusString = @"The logo has been downloaded successfully";
            }
        }
        [self.delegate resultadoConexionGet:statusString result:result];
```

Figura 37. Código método receiveDidStopWithStatus:. Consulta GET

En el método *receiveDidStopWithStatus:* de la clase *EmpresasConnection* se envía un mensaje *resultadoConexionGet:result:* al controlador de vista que inicializó la conexión. Se le pasa como parámetros una cadena de texto que se mostrará al usuario y el resultado de la conexión (véase Fig. 37).

En la Fig. 38 se muestra el código del método *resultadoConexionGet:result:*. Si la conexión ha sido un éxito se registra la nueva empresa. En caso contrario se informa al usuario del error.

```
-(void)resultadoConexionGet:(NSString *)statusString result:(BOOL)result {
    self.esperandoRespuesta = false;


    if (result) {

        [self.delegate agregaEmpresaP:self withEmpresa:self.empresa
password:self.password.text index:0];

    }else {
        //Se muestra la ventana con la información para el usuario
        UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Error!"
                                                        message:statusString
                                                       delegate:nil
                                              cancelButtonTitle:nil
                                              otherButtonTitles:@"Done", nil];
        [alert setAlertViewStyle:UIAlertViewStyleDefault];
        [alert show];
    }
}
```

Figura 38. Código método resultadoConexiónGet:result:

3.3.3.2.1 Petición de tipo Get

En la Fig. 39 se muestra el código utilizado para establecer una comunicación con el servidor y solicitar el logo de una de las empresas. Pertenece al método *startReceiveGet*. Esta operación se realiza cada vez que se registra una nueva empresa para solicitar la imagen que contiene el logo de la empresa.

```
//Abro un stream para poder guardar los paquetes http recibidos
self.filePath = [self pathForTemporaryFileWithPrefix:@"Get"];


self.fileStream = [NSOutputStream outputStreamToFileAtPath:self.filePath
append:NO];


[self.fileStream open];


//Conexion URL. Se crea la petición GET (Para solicitar el logo)
NSURL *url = nil;

//PARA IBERIA
if([self.host isEqualToString:@"www.iberia.com"]) {
     url NSURL URLWithString: @"http://www.iberia.es/ibcomv3/images/ldc/logo_home.png"];

NSURLRequest *getRequest = [NSURLRequest requestWithURL:url
cachePolicy:NSURLRequestUseProtocolCachePolicy timeoutInterval:60.0];

//Se crea la conexión con la petición
self.Connection = [[NSURLConnection alloc] initWithRequest:getRequest
delegate:self];

if (self.connection) {
        //Si la conexión es correcta
} else {
        //Inform the user that the connection failed.
}
```

Figura 39. Código método startReceiveGet

Es necesario abrir un stream de escritura para guardar la respuesta del servidor en un fichero. La ruta del fichero se obtiene de la función *pathForTemporaryFileWithPrefix* que devuelve la ruta de un directorio temporal para el usuario actual.

La consulta es un objeto *NSURLRequest* que se crea con la URL a la que se quiere realizar la petición. Finalmente se inicializa la conexión con la consulta deseada.

3.3.3.2.2 Métodos delegados

También es necesario implementar como mínimo los siguientes métodos delegados: connection:didReceiveResponse:, connection:didReceiveData:, connection:didFailWithError: y connectionDidFinishLoading:.

Cuando el servidor tiene suficientes datos para crear una respuesta, la recibe el método delegado *connection:didReceiveResponse:*. Este método puede examinar el origen de la respuesta, determinar el tipo y el tamaño de los datos que contiene (véase Fig. 40).

```
- (void)connection:(NSURLConnection *)connection didReceiveResponse:(NSURLResponse
*)response
{
//Comprobación del contenido de la respuesta http (de la cabecera)
    NSHTTPURLResponse *httpResponse;
    NSString *contentTypeHeader;

    httpResponse = (NSHTTPURLResponse *) response;

    if ((httpResponse.statusCode / 100) != 2) {
        [self stopReceiveWithStatus:[NSString stringWithFormat:@"HTTP error %zd",
(ssize_t) httpResponse.statusCode]];
    } else {
        contentTypeHeader = [httpResponse MIMEType];
        if (contentTypeHeader == nil) {
            [self stopReceiveWithStatus:@"No Content-Type!"];
        } else if ( [contentTypeHeader isEqual:@"text/plain"]) {
            self.post = true;
            NSLog(@"HTML Response OK3.");
        } else if ( [contentTypeHeader isEqual:@"text/html"]) {
            self.post = true;
            NSLog(@"HTML Response OK2.");
        } else if([contentTypeHeader isEqual:@"text/html; charset=UTF-8"]) {
            self.post = true;
            NSLog(@"HTML Response OK1.");
        }else if ([contentTypeHeader isEqual:@"image/png"] || [contentTypeHeader
isEqual:@"image/jpeg"] ) {
            self.logo = true;
            self.get = true;
            NSLog(@"Logo Response OK.");
        }else {
            [self stopReceiveWithStatus:[NSString stringWithFormat:@"Unsupported
Content-Type (%@)", contentTypeHeader]];
        }
    }
}
```

Figura 40. Codigo método connection:didReceiveResponse:

La respuesta es un objeto de tipo *NSURLResponse*, pero la respuesta que se espera es un paquete http, por lo que para acceder a los campos de la cabedera de una paquete http es necesario hacer un casting para poder trabajar con un objeto de tipo *NSHTTPURLResponse*. Se puede hacer el casting porque la clase *NSHTTPURLResponse* es una clase hija de la clase *NSURLResponse*. Una vez que se ha hecho el casting se identifica el tipo de datos de la respuesta del servidor.

El método *connection:didReceiveData:* es llamado periódicamente cuando se han recibido datos. Es el responsable de guardar los nuevos datos recibidos. Se muestra el código a continuación. Los datos se reciben en un objeto de tipo *NSData* y se guardan en el fichero codificados en formato UTF-8 a través del stream abierto anteriormente (véase Fig. 41).

```
- (void)connection:(NSURLConnection *)theConnection didReceiveData:(NSData *)data
//Tratamiento de los datos de la respuesta http
{
    NSInteger       dataLength;
    const uint8_t   *dataBytes;
    NSInteger       bytesWritten;
    NSInteger       bytesWrittenSoFar;
    NSString        *recibido;

    if(self.get || self.post) {
        recibido = [[NSString alloc] initWithData:data
encoding:NSUTF8StringEncoding];

        assert(theConnection == self.connection);
        dataLength = [data length];
        dataBytes  = [data bytes];

        bytesWrittenSoFar = 0;
        do {
            //Guardo el código html (o la imagen) recibido en el stream fileStream
            bytesWritten = [self.fileStream write:&dataBytes[bytesWrittenSoFar]
maxLength:dataLength - bytesWrittenSoFar];
            assert(bytesWritten != 0);
            if (bytesWritten == -1) {
                [self stopReceiveWithStatus:@"File write error"];
                break;
            } else {
                bytesWrittenSoFar += bytesWritten;
            }
        } while (bytesWrittenSoFar != dataLength);

        if(!self.aux) {
            self.parcialLength = dataLength;
            self.aux = true;
        }
        self.totalLength += dataLength;
    }
}
```

Figura 41. Código método connection:didReceiveData:

Si se produce algún error durante la conexión, el delegado recibe un mensaje *connection:didFailWithError:*. El objeto NSError que se le pasa como parámetro especifica los detalles del error. También proporciona la URL de la consulta que ha fallado. En este caso se limita a comunicar al usuario que la conexión ha fallado (véase Fig. 42).

```
- (void)connection:(NSURLConnection *)theConnection didFailWithError:(NSError
*)error
{
    [self stopReceiveWithStatus:@"Connection failed"];
}
```

Figura 42. Código método connection:didFailWithError:

Finalmente, si la conexión tiene éxito el delegado recibe un mensaje *connectionDidFinishLoading:*.

El delegado no recibirá más mensajes, por lo que se puede cerrar la conexión. En la aplicación, una vez finalizada con éxito la conexión, en el caso de que se haya realizado una consulta de tipo POST, se llama a la función *findInHtmlWith:desplazamiento:primerCaracter:* para obtener, del código HTML recibido, los datos de interés.

3.3.3.3 Autenticación

Volviendo a la figura y con el ejemplo de la clase *AddViewController*, con la diferencia de que en este apartado se va a tratar la consulta de tipo POST.

Cuando el usuario pulsa el butón *Check* para validar los datos de la empresa se inicia la conexión con el servidor. En la Fig. 43 se muestra el código de la acción *check:* que se realiza en la clase *EmpresasAddViewController*.

```
- (IBAction)check:(id)sender {
    if   ([self.nombreEmpresa.text   length]   &&   [self.urlWeb.text   length]   &&
[self.username.text length] && [self.password.text length]) {
        //Si el usuario ha rellenado todos los campos
        if (!self.esperandoRespuesta) {
            self.esperandoRespuesta = true;
            //Rellenan los campos de la empresa con la información introducida por
el usuario
            self.empresa.nombreEmpresa = self.nombreEmpresa.text;
            self.empresa.urlWeb = self.urlWeb.text;
            self.empresa.username = self.username.text;

            //Se crea la conexión
            self.conexion= [[EmpresasConnection alloc]initWithEmpresa:self.empresa
password:self.password.text delegate:self];

            //Se realiza la petición
            self.post = true;
            [self.conexion startReceivePost];
        }
    }else {
        //Se avisa al usuario de que tiene que rellenar todos los campos
    }
}
```

Figura 43. Código acción check

Igual que para conexiones de tipo GET, se inicializa la propiedad *conexión* mediante el método *initWithEmpresa:password:delegate:*. A través de este mensaje se le pasan los datos de la empresa introducidos por el usuario y se le indica cual es el controlador de vista que está iniciando la conexión.

Para autenticarse se envía una consulta de tipo POST a través del método *startReceivePost* perteneciente también a la clase *EmpresasConnection.*

Una vez finalizada la conexión, el método *receiveDidStopWithStatus:* de la clase *EmpresasConnection* envía un mensaje *resultadoConexionPost:result:* al controlador de vista que inicializó la conexión. Se le pasa como parámetros una cadena de texto, que se mostrará al usuario, y el resultado de la conexión (véase Fig. 44).

```
if (self.post) {
        //Se ha enviado una petición de autenticación
        if (self.empresa.puntos == nil) {
            result = false;
            if (statusString == nil) {
                statusString = @"Can not access your account, check that the data
entered is correct";
            }
        }else {
            result = true;
            if (statusString ==nil) {
                statusString = @"The data is correct. You can register the
company";
            }
        }
        [self.delegate resultadoConexionPost:statusString result:result];
  }
```

Figura 44. Código método receiveDidStopWithStatus:. Consulta POST

En la Fig. 45 se muestra el código del método *resultadoConexionPost:result:*. Si la conexión ha sido un éxito se habilita el botón *Done* y se le comunica al usuario de que puede registrar la empresa. En caso contrario se le informa del error.

```
- (void)resultadoConexionPost:(NSString *)statusString result:(BOOL)result {
    self.esperandoRespuesta = false;
    //Una vez obtenida la respuesta se informa al usuario del resutlado
    NSLog(@"LLegó: %d,%d",self.post,result);
    if (self.post) {
        if (!result) {
            //Se muestra la ventana con la información para el usuario
            UIAlertView *alert = [[UIAlertView alloc]
                                  initWithTitle:@"Error!"
                                        message:statusString
                                       delegate:nil
                              cancelButtonTitle:nil
                              otherButtonTitles:@"Done", nil];
            [alert setAlertViewStyle:UIAlertViewStyleDefault];
            [alert show];
        }else {
            //Los datos son correctos, ya se puede registrar la empresa y hacer la
petición para obtener el logo
            self.doneBarButtonItem.enabled = true;

            //Se muestra la ventana con la información para el usuario
            UIAlertView *alert=[[UIAlertView alloc]
              initWithTitle:@"Congratulations!"
                    message:@"The data is correct. You can register the company"
                   delegate:nil
          cancelButtonTitle:nil
          otherButtonTitles:@"Done", nil];
            [alert setAlertViewStyle:UIAlertViewStyleDefault];
            [alert show];

            self.post = false;
        }
    }
}
```

Figura 45. Código método resultadoConexiónPost:result:

3.3.3.3.1 Petición de tipo POST

En la Fig. 46 se muestra parte del código del método *startReceivePost.* De la misma forma que para una consulta de tipo GET, es necesario abrir un stream para guardar los datos recibidos por el servidor. La ruta del fichero se obtiene de la función *pathForTemporaryFileWithPrefix:*.

```
    //Inicializo variables
     if (self.filePath != nil) {
        self.filePath = nil;
    }
    self.totalLength = 0;

    //Abro un stream para poder guardar los paquetes http recibidos
    self.filePath = [self pathForTemporaryFileWithPrefix:@"Post"];
    assert(self.filePath != nil);
    self.fileStream   =   [NSOutputStream   outputStreamToFileAtPath:self.filePath
append:NO];
    assert(self.fileStream != nil);
    [self.fileStream open];

    //Conexion URL. Se crea la petición POST (Loguearse)
    NSURL *url = [NSURL URLWithString:self.empresa.urlWeb];
    NSMutableURLRequest *postRequest = [NSMutableURLRequest requestWithURL:url];
    self.host = [url host];

    //Si no han pasando la clave la obtengo del llavero
    if (self.password == nil) {
        NSString            *key            =            [EmpresasKeychainWrapper
keychainStringFromMatchingIdentifier:PIN_SAVED];
        self.password                    =                    [EmpresasKeychainWrapper
obtenerClaveWithNombreEmpresa:self.empresa.nombreEmpresa key:key];
    }
    //Se crea el cuerpo de la petición POST
    NSString *bodyData;
    //PARA IBERIA
    if([self.host isEqualToString:@"www.iberia.com"]) {
        bodyData = @"username=";
        bodyData = [bodyData stringByAppendingString:self.empresa.username];
        bodyData = [bodyData stringByAppendingString:@"&password="];
        bodyData = [bodyData stringByAppendingString:self.password];
    }else {
        NSLog(@"Host desconocido");
    }
    // Set the request's content type to application/x-www-form-urlencoded
    [postRequest                        setValue:@"application/x-www-form-urlencoded"
forHTTPHeaderField:@"Content-Type"];

    // Designate the request a POST request and specify its body data
    [postRequest setHTTPMethod:@"POST"];
    [postRequest     setHTTPBody:[NSData     dataWithBytes:[bodyData     UTF8String]
length:[bodyData length]]];

    //Se crea la conexión con la petición
    self.Connection   =   [[NSURLConnection   alloc]   initWithRequest:postRequest
delegate:self];

    if (self.connection) {
        //Si la conexión es correcta
    } else {
        // Inform the user that the connection failed.
    }
```

Figura 46. Código método startReceivePost

Se crea un objeto de tipo *NSURL* que contiene la URL a la que se va a realizar la consulta. La consulta, de tipo *NSMutableURLRequest*, se crea mediante el método *requestWithURL:* pasándole como parámetro el objeto *NSURL*.

En los datos de la consulta tiene que contener el nombre de usuario y la contraseña con la que se pretende autenticar al usuario en el servidor. Como la clase *EmpresasConnection* tiene acceso al llavero, se puede obtener la clave de la empresa. Estos métodos se explicarán en el siguiente apartado que trata sobre el almacenamiento de claves.

3.3.3.3.2 Métodos delegados

Una consulta de tipo *NSURLRequest* a menudo puede encontrarse con un desafío de autenticación o puede necesitar credenciales para conectarse al servidor.

Si un objeto *NSURLRequest* requiere autenticación, el delegado del objeto *NSURLConnection* asociado a la consulta recibe primero un mensaje *connection:canAuthenticateAgainstProtectionSpace:*. Este método permite al delegado analizar las propiedades del servidor, sus protocolo y método de autenticación antes de intentar autenticarse.

```
-(BOOL)connection:(NSURLConnection *)connection
canAuthenticateAgainstProtectionSpace:(NSURLProtectionSpace *)protectionSpace {

    if([protectionSpace.authenticationMethod
isEqualToString:NSURLAuthenticationMethodServerTrust])
    {
        NSLog(@"NSURLAuthenticationMethodServerTrust");
        return YES;
    }
    else if([protectionSpace.authenticationMethod
isEqualToString:NSURLAuthenticationMethodHTTPBasic])
    {
        NSLog(@"Can Auth Basic Requestes!");
        return YES;
    }
    NSLog(@"Cannot Auth!");
    return NO;
}
```

Figura 47. Código método connection:canAuthenticateAgainstProtectionSpace:

Tal y como se muestra en la Fig. 47, en el código se tratan dos métodos de autenticación. El método de autenticación básica o *NSURLAuthenticationMethodHTTPBasic* y el método de autenticación con credenciales o *NSURLAuthenticationMethodServerTrust*.

El método de autenticación básica requiere un nombre de usuario y una contraseña. Mientras que el método de autenticación de confianza requiere de unos credenciales proporcionados por el espacio de protección del desafío de autenticación.

Si no se dispone de la información necesaria para la autenticación el delegado recibe un mensaje *connection:didReceiveAuthenticationChallenge:*. Para que la conexión continúe, en este método se debe de proporcionar la información o los credenciales necesarios para la autenticación. También existen las opciones de intentar continuar sin credenciales o cancelar la autenticación.

```
-(void)connection:(NSURLConnection *)connection
didReceiveAuthenticationChallenge:(NSURLAuthenticationChallenge *) challenge {

    if ([challenge.protectionSpace.authenticationMethod
isEqualToString:NSURLAuthenticationMethodServerTrust])
    {
        //Obtengo el host
        self.host = challenge.protectionSpace.host;

        NSLog(@"Trust Challenge Requested!");

        //Usar credencial para la autenticación
        [challenge.sender useCredential:[NSURLCredential
credentialForTrust:challenge.protectionSpace.serverTrust]
forAuthenticationChallenge:challenge];

    }else if([challenge.protectionSpace.authenticationMethod
isEqualToString:NSURLAuthenticationMethodHTTPBasic])
    {
        NSLog(@"HTTP Auth Challenge Requested!");
        if ( [challenge previousFailureCount] == 0 ) {
            NSError *er = [challenge error];
            NSLog(@"Error de autenticación: %@", er);

            NSURLCredential *credential = [[NSURLCredential alloc]
initWithUser:self.empresa.username password:self.password.text
persistence:NSURLCredentialPersistenceForSession];

            [[challenge sender] useCredential:credential
forAuthenticationChallenge:challenge];
        }else {
            [challenge.sender cancelAuthenticationChallenge:challenge];
            NSLog(@"Error de autenticación");
        }
    }
}
```

Figura 48. Código método connection:didReceiveAuthenticationChallange:

La instancia del objeto *NSURLAuthenticationChallange* pasado al método como parámetro contiene información sobre lo que desencadenó el desafío de autenticación, el número de intentos hasta el momento para superar el desafío, los credenciales que se usaron, los que se requieren y el emisor del desafío.

Se puede observar en la Fig. 48 como se tratan los dos métodos de autenticación. En ambos se usa el método *useCredential:forAuthenticationChallenge:* para responder al desafío de autenticación.

En el método de autenticación básica se crean los credenciales, *NSURLCredential*, mediante el método *initWithUser:password:persistence:* pasándole como parámetros el nombre de usuario y la contraseña.

Para la autenticación de confianza se usa el método *credentialForTrust:challenge.protectionSpace.serverTrust*, obteniéndose los credenciales del espacio de protección del propio desafío de autenticación.

Resuelto el desafío de autenticación se continúa con la conexión. En el caso de que se trate de otro método de autenticación se cancela la autenticación a través del mensaje *cancelAuthenticationChallenge:challenge:*.

3.3.3.4 Almacenamiento de claves, servicios de llavero

Proporciona almacenamiento seguro de contraseñas, llaves y certificados para uno o más usuarios. Un llavero es un contenedor encriptado que solo es accesible a través de su aplicación [KSPG2012].

Cada llavero puede contener cualquier número de elementos y cada elemento contiene datos además de un conjunto de atributos. Los atributos asociados a un elemento del llavero dependen de la clase del elemento.

La clave del usuario y las contraseñas de las diferentes empresas registradas se almacenarán en el llavero de la aplicación.

La encargada de las tareas relacionadas con el servicio llavero, es la clase *EmpresasKeychainWrapper*. En esta clase todos los métodos están definidos como métodos de clase, es decir, que no es necesario crear un objeto de la clase para poder llamar a estos métodos.

3.3.3.4.1 Clave de usuario

El usuario se registra con un nombre de usuario y una contraseña la primera vez que entra en la aplicación. El controlador de vista EmpresasRootViewController es el encargado de pedirle al usuario la contraseña para acceder a la aplicación o mostrarle el formulario para registrarse.

El método *textFieldDidEndEditing:* es un método delegado que permite examinar el texto introducido en los campos de texto. Se usa para comprobar si la clave introducida es correcta y para registrar al usuario (véase Fig. 49).

```
- (void)textFieldDidEndEditing:(UITextField *)textField
{
    switch (textField.tag) {
        case kTextFieldPIN: //El usuario ya esta registrado, tiene que introducir
la clave para entrar en la aplicación
            if ([textField.text length] > 0) {
                NSUInteger fieldHash = [textField.text hash];

                if ([EmpresasKeychainWrapper
compareKeychainValueForMatchingPIN:fieldHash]) {
                    self.pinValidated = YES;
                } else {
                    self.pinValidated = NO;
                }
            }
            break;
        case kTextFieldName: //Registro del nombre de usuario de la aplicación
            if ([textField.text length] > 0) {
                [[NSUserDefaults standardUserDefaults] setValue:textField.text
forKey:USERNAME];
                [[NSUserDefaults standardUserDefaults] synchronize];
            }
            break;
        case kTextFieldPassword: //Registro de la contraseña de la aplicación.
            if ([textField.text length] > 0) {
                NSUInteger fieldHash = [textField.text hash];
                NSString *fieldString = [EmpresasKeychainWrapper
securedSHA256DigestHashForPIN:fieldHash];
```

```
                if ([EmpresasKeychainWrapper createKeychainValue:fieldString
forIdentifier:PIN_SAVED]) {
                    [[NSUserDefaults standardUserDefaults] setBool:YES
forKey:PIN_SAVED];
                    [[NSUserDefaults standardUserDefaults] synchronize];
                }
            }
            break;
        default:
            break;
    }
}
```

Figura 49. Código método textFieldDidEditing:

Se diferencian tres estados distintos, el primero es cuando el usuario ya está registrado y se le pide la contraseña para acceder a la aplicación. Una vez introducida la clave se cifra con el algoritmo SHA y se compara con la contraseña que se guardó en el llavero cuando el usuario se registró, si coinciden se permite el acceso a la aplicación. Para ello primero se calcula el valor hash de la clave introducida mediante el método *hash*. Y para comparar las claves cifradas se usa el método *compareKeychainValueForMatchingPIN:* de la clase *EmpresasKeychainWrapper*.

En el segundo y tercer estado es cuando el usuario acceder por primera vez a la aplicación. En el segundo caso se guarda el nombre de usuario y se indica que el usuario ya está registrado, para realizar estas tareas se usa la clase *NSUserDefautls*.

Esta clase proporciona una interfaz para interactuar con los valores predeterminados del sistema. Este permite personalizar su comportamiento para configurar las preferencias de usuario. Por ejemplo el usuario puede determinar con que frecuencia se guardan de forma automática los documentos. También permite que las aplicaciones guarden preferencias, asignando valores a un conjunto de parámetros en la base de datos del usuario por defecto.

El método *synchronize* guarda, en la base de datos del usuario predeterminado, los cambios realizados relacionados con la clase *NSUserDefaults*.

Por último en el tercer caso se guarda la contraseña cifrada mediante el algoritmo SHA (Secure Hash Algorithm), para ello se obtiene el valor hash de la clave y se pasa como parámetro al método *securedSHA256DigestHashForPIN:*. Este método devuelve un objeto de tipo *NSString* con la clave encriptada. Mediante el método *createKeychainValue:forIdentifier:* la clave se guarda en el llavero pasándole también el identificador como parámetro. Para una mayor seguridad las claves siempre se guardan cifradas, nunca en texto plano. Todos estos métodos también pertenecen a la clase EmpresasKeychainWrapper.

3.3.3.4.2 Claves de las empresas registradas

Al registrar una nueva empresa, mediante el controlador de vista *AddViewController*, la clave se guarda en el llavero a través del método *guardarClave:withNombreEmpresa:key:*. Como se puede observar hay que pasarle como parámetros la clave, el nombre de la empresa y una llave. En la Fig. 50 se muestra el código que pertenece al método *agregaEmpresaWithEmpresa:password:index:,* implementado en la clase *EmpresasMasterViewController,* ya comentado anteriormente.

```
if(password) {

        [EmpresasKeychainWrapper guardarClave:password
withNombreEmpresa:nuevaEmpresa.nombreEmpresa key:key];

}
```

Figura 50. Código guardar clave

El nombre de la empresa va a ser el identificador para obtener la clave del llavero cuando se solicite. Para cifrar la clave se usa como llave la contraseña del usuario que se ha registrado en la aplicación. Las contraseñas de las diferentes empresas se van a cifrar mediante el algoritmo AES (Advanced Encryption Standard) ya que aunque las claves se guarden cifradas, se necesita obtener la contraseña en texto plano para poder autenticarse en los distintos servidores. En la Fig. 51 se muestra el código de la función *guardarClave:withNombreEmpresa:key:* de la clase *EmpresasKeychainWrapper*.

```
+(void)guardarClave:(NSString *)password withNombreEmpresa:(NSString
*)nombreEmpresa key:(NSString *)key {

    if ([password length] > 0) {

        NSString *secret = password;
        NSData *plain = [secret dataUsingEncoding:NSUTF8StringEncoding];
        NSData *cipher = [self AES256EncryptWithKey:key clave:plain];

        if ([self createKeychainData:cipher forIdentifier:nombreEmpresa]) {
            NSLog(@"** Key saved successfully to Keychain!!");
        }

    }
}
```

Figura 51. Código método guardarClave:withNombreEmpresa:key:

Para cifrar la clave siguiendo el algoritmo AES se usa el método *AES256EncryptWithKey:clave:*. Una vez cifrada se guarda en el llavero a través de la función *createKeychainData:forIdentifier:*.

Cuando se quiera acceder a una empresa registrada hay que acceder al llavero. Pero para acceder a la clave, es necesario obtener primero la llave. Esta se pasa como parámetro a la función *obtenerClaveWithNombreEmpresa:key:*, junto con el nombre de la empresa para obtener la clave en texto plano.

El código de la Fig. 52 pertenece al método *startReceivePost* de la clase *EmpresasConnection*, también mencionado anteriormente. En el caso de que no se le haya pasado la clave de la empresa la busca en el llavero.

```
//Si no han pasando la clave la obtengo del llavero
    if (self.password == nil) {
        NSString *key = [EmpresasKeychainWrapper
keychainStringFromMatchingIdentifier:PIN_SAVED];
        self.password = [EmpresasKeychainWrapper
obtenerClaveWithNombreEmpresa:self.empresa.nombreEmpresa key:key];
    }
```

Figura 52. Código obtener clave

En el método obtenerClaveWithNombreEmpresa:key:, se descifra la clave mediante el método *AES256DecryptWithKey:clave:*. Este método devuelve un objeto de tipo *NSData*, pero se puede pasar fácilmente a una cadena de caracteres (*NSString*) aplicando una codificación UTF-8 (véase Fig. 53).

```
+(NSString *)obtenerClaveWithNombreEmpresa:(NSString *)nombreEmpresa key:(NSString
*)key {

    NSData *data = [self searchKeychainCopyMatchingIdentifier:nombreEmpresa];

    NSData *claveDesencriptada = [self AES256DecryptWithKey:key clave:data];

    const char *fieldChar = [[[NSString alloc] initWithData:claveDesencriptada
encoding:NSUTF8StringEncoding] UTF8String];

    NSString *fieldString = [NSString stringWithUTF8String:fieldChar];

    return fieldString;
}
```

Figura 53. Código obtenerClave:withNombreEmpresa:key:

4 Conclusión

En resumen, para este proyecto, se ha estudiado la situación actual, las tendencias y las tecnologías empleadas sobre como acceden los Smartphones a la información de Internet.

Se ha realizado un caso práctico desarrollando una aplicación para el sistema operativo iOS. Una primera parte ha consistido en el aprendizaje del lenguaje de programación (Objetive-C) y el manejo de las herramientas utilizadas.

También se han analizado los tipos de mensajes que se envían al realizar una consulta a un servidor web, y al autenticarse. Que protocolos se emplean y el formato en el que se envían y reciben los datos.

Para el almacenamiento seguro de claves, se ha profundizado en los dos algoritmos de cifrado mencionados anteriormente para su posterior implementación.

Como opinión personal quiero destacar la buena documentación que proporciona Apple Inc. para el desarrollo de aplicaciones a través de la sección iOS Devoloper Library disponible en su página web. Siguiendo la documentación el desarrollador va profundizando desde la información más general hasta la más específica según sus necesidades.

5 Referencias

[AES13] Advanced Encryption Standard, Wikipedia, 2013.
[SHA13] Secure Hash Algorithm, Wikipedia, 2013.
[IHIG12] iOS Human Interface Guidelines, Apple Inc., 2012.
[CDPG12] Core Data Programming Guide, Apple Inc., 2012.
[CDME12] Core Data Model Editor Help, Apple Inc., 2012.
[CFG10] Cocoa Fundamentals Guide, Apple Inc., 2010.
[INO13] Networking Overview, Apple Inc., 2013.
[ULSP10] URL Loading System Programming Guide, Apple Inc., 2010.
[KSPG12] Keychain Services Programming Guide, Apple Inc., 2012.
[FOS13] Firefox OS 1.0.1 Notes, Mozilla, 2013.
[CAPT13] Evolución de la Web en el mundo móvil, Manuel Vida en Blog Captative, 2013.